Einführung in das PVS-System

Maschinelles Beweisen – WS 06/07

16. November 2006

Was ist PVS?

- interaktiver Theorembeweiser
- entwickelt am Computer Science Laboratory,
  SRI International, Menlo Park (Kalifornien), von S. Owre, N. Shankar, J. Rushby, u.a.
- erste Version 1992, aktuell: PVS 3.3 (release candidate)
- ausdrucksstarke Spezifikationssprache
- mächtige Basisinferenzen
- Entscheidungsprozeduren
- einfache Sprache zur Definition von Beweisstrategien

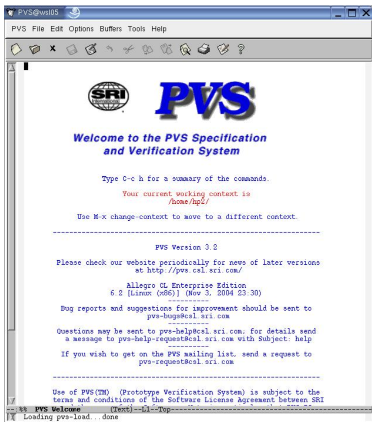
Installation

- Im Linux-Pool:
  - `use pvs`
  - Aufruf mit `pvs`
- Auf eigenem Rechner:
  - `http://pvs.csl.sri.com` oder `http://www.informatik.uni-ulm.de/ki/PVS/mirror.html`
  - Dateien in Installationsverzeichnis entpacken, dort das Skript `./bin/relocate` ausführen
  - evtl. Installationsverzeichnis in `PATH`-Variable aufnehmen, oder symbolischer Link in `/usr/local/bin/` auf `pvs`
  - Aufruf mit `pvs`

Das PVS-System

- PVS läuft als inferior Lisp-Prozess in einer EMACS-Oberfläche.
- Online-Tutorial zu Emacs mit `C-h t`.
  Online-Dokumentation zu Emacs mit `C-h i`.
  Online-Übersicht der PVS-Kommandos mit `C-c h`.
- Vor dem ersten Starten: Anlegen eines eigenen Verzeichnisses, in dem PVS-Dateien aufbewahrt werden.
- Aufruf von PVS in dem entsprechenden Verzeichnis. Auf Frage nach Anlegen eines neuen Kontextes (bei erstmaliger Verwendung) mit `yes` antworten.
- Verlassen von PVS mit `C-x C-c`.

Arbeitszyklus



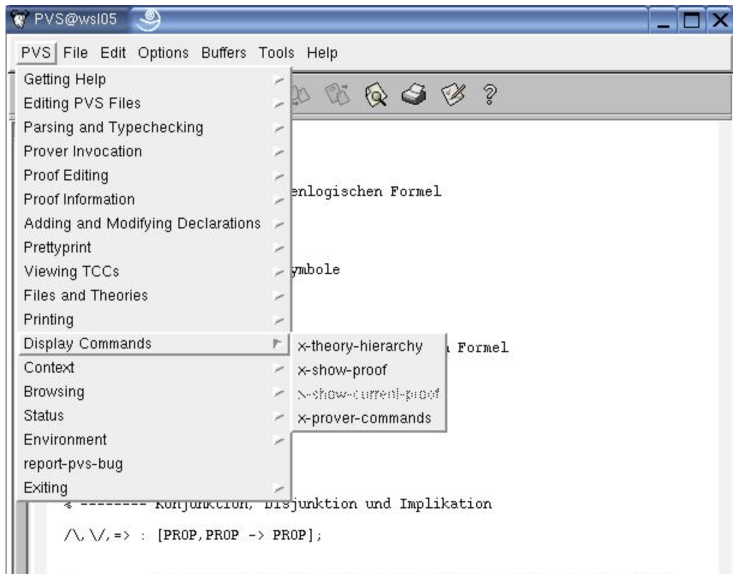
- in ein Arbeitsverzeichnis (PVS: context) wechseln:
  - `M-x cc`, Verzeichnisnamen eingeben
- neue PVS-Theorie erstellen:
  - `M-x nf`, Theorienamen eingeben
- bestehende Theorie laden:
  - `C-c C-f`, Theorienamen eingeben
- Spezifikation, Typprüfung, Beweis
  - im folgenden beschrieben ...
- PVS beenden:
  - `C-x C-c`, zur Bestätigung `y` eingeben

Die wichtigsten Emacs-Kommandos für PVS

- PVS Hilfe: `C-c h`
  - ...zur Spezifikationssprache: `C-c C-h l`
  - ...zum Beweiser: `C-c C-h p`
  - ...zu Beweiserkommandos `C-c C-h c`
  - ...zu Emacs-Kommandos im Beweiser `C-c C-h e`
- Theorie laden: `C-c C-f` und speichern: `C-x C-s`
- Typprüfung: `M-x tc` oder `C-c t`
  - ...mit Beweis der TCCs: `M-x tcp`
- Formel beweisen: `M-x pr` oder `C-c p`
  - ...mit Beweisbaum: `M-x x-prove`, `M-x xpr`, oder `C-c C-p x`
  - ...schrittweise: `M-x step-prove`, `M-x prs` oder `C-c C-p s`
  - ...schrittweise mit Beweisbaum `M-x x-step-prove`, `M-x xsp` oder `C-c C-p X`
- Typkorrektheitsbedingungen
  - ...anzeigen: `M-x show-tccs` oder `C-c C-q s`
  - ...beweisen: `M-x prove-tccs-theory` oder `M-x prtt`
- Beweise bearbeiten: `M-x edit-proof`
  - ...installieren (speichern): `M-x install-proof` oder `C-c C-i`

PVS Menüs

Viele der PVS-Kommandos sind auch über Menüs erreichbar



- Cursor auf zu beweisende Formel, Kommando `M-x pr` oder `C-c p`
- weiterer Emacs-Buffer wird geöffnet:

```
--:--  aussagenlogik.pvs      (PVS :ready)--L52--Bot------------------------------
Starting pvs-allegro6.2 -qq ...
Allegro CL Enterprise Edition
6.2 [Linux (x86)] (Nov 3, 2004 23:30)
Copyright (C) 1985-2002, Franz Inc., Berkeley, CA, USA.  All Rights Reserved.

This dynamic runtime copy of Allegro CL was built by:
   [TC8101] SRI International

;; Optimization settings: safety 1, space 1, speed 3, debug 1.
;; For a complete description of all compiler switches given the
;; current optimization settings evaluate (explain-compiler-settings).
;;---
;; Current reader case mode: :case-sensitive-lower
pvs(1):
pvs(2):

thm1 :

  |-------
{1}   valid(A /\ B => A)

Rule? █
--:**  *pvs*            (ILISP :ready)--L22--All------------------------------------
aussagenlogik typechecked in 0.04s: No TCCs generated; 2 msgs
```

- `Rule?`-Prompt erwartet Beweiserkommandos

Hilfequellen

- PVS homepage: `http://pvs.csl.sri.com`
- Dokumentation:
  `http://pvs.csl.sri.com/documentation.shtml`
  - Systembeschreibung
  - Language Reference
  - Prover Guide
  - Tutorials, viele Papers, etc.
- Hilfesystem von PVS: `C-c h`
- unsere Kurzanleitung(en)
- Kommilitonen ...

Umgang mit PVS

Die Arbeit mit PVS läuft grob in drei Phasen ab:

1. **Spezifizieren:**
   Deklaration von Namen für Typen und Variablen; Definition von Typen, Konstanten, Funktionen; Deklaration von Formeln (Axiome, Lemmata, Theoreme).
2. **Typprüfen:**
   Überprüfen der statischen Semantik der Spezifikation. Typkorrektheit kann von sogenannten *Typkorrektheitsbedingungen* (type correctness conditions, TCCs) abhängen.
3. **Beweisen:**
   Beweis der Typkorrektheitsbedingungen, Beweis der deklarierten Lemmata und Theoreme.

Spezifikationen

- Theorien bilden die Strukturierungseinheiten für Spezifikationen in PVS. Theorien haben im wesentlichen folgende Form:

```
<Name> [<formale Parameter>] : THEORY
BEGIN
  ASSUMING
    <Assumptions>
  ENDASSUMING
  IMPORTING <Theorien ...>
  <Deklarationen, Definitionen, Formeln>
END
```

- Anlegen einer neuen Theorie mit `M-x nf`, danach Angabe eines Namens.
- Speichern von Theorien mit `C-x C-s`. Nach wichtigen Interaktionen werden die Theorien automatisch gespeichert.
- Bereits bestehende Dateien können mit `C-c C-f` geladen werden.

Elemente von PVS Theorien: Typen

- Deklaration mit `name : TYPE`
- vordefinierte Typen: `bool`, `nat`, `int`, `real`, `list`, ...
  siehe Prelude-Datei (`M-x view-prelude-file`).
- Definition neuer Typen:

Funktionstypen:	`fctn   : TYPE = [int, int -> int]`
Tupel:	`triple : TYPE = [nat, fctn, list[int]]`
Records:	`record : TYPE = [# a:A, b:B #]`
Subtypen:	`posnat : TYPE = {x:nat | x>0}`

Elemente von PVS Theorien: Konstanten

- Definitionen durch Angabe von Name, Type und Wert

```
five   : nat         = 5
pair   : [nat,bool] = (five,TRUE)
primes : list[nat]  = (: 2, 3, 5 :)
                  % = cons(2,cons(3,cons(5,null)))
```

- Funktionskonstanten:

```
inc1         : [nat -> nat] = LAMBDA (n:nat): n+1
inc2(n:nat) : nat           = n+2
```

Elemente von PVS Theorien: Variablen

- Einführung durch Angabe von Bezeichner und Typ

```
n : VAR nat
```

- zur Abkürzung von Funktionsdefinitionen

```
inc2(n:nat) : nat = n+2
```

oder

```
n : VAR nat
inc2(n) : nat      = n+2
```

- zur impliziten All-Quantifizierung von Formeln

```
n : VAR nat
thm : THEOREM
  n < n + 1
```

bedeutet:

```
thm : THEOREM
  FORALL (n:nat): n < n + 1
```

Elemente von PVS Theorien: Formeln

- Formeln sind Ausdrücke vom Typ `bool`.
- vordefinierte Junktoren: `NOT, AND, OR, IMPLIES, IFF`
- Prädikate sind Boolesche Funktionen: `P : [T -> bool]`, Abk.: `P : pred[T]`
- Quantifizierte Ausdrücke:
  `EXISTS (x:T): P(x)`, `FORALL (y:T): P(y)`
- In eine Theorie werden Formeln eingeführt durch:
  <Name> : <Bezeichnung> <Formel>
- Je nach Bezeichnung werden die Formel unterschiedlich behandelt:
  - `AXIOM`: Formel wird als wahr angenommen und kann bei Beweisen als Voraussetzung einfließen.
  - `LEMMA, THEOREM` (etc.): Für die Formel muss (irgendwann) ein Beweis geliefert werden.
- Reihenfolge der Deklarationen: Axiome, Lemmata, etc. können erst in Beweisen der nachfolgend deklarierten Formeln verwendet werden

Beispieltheorie

```
gruppe : THEORY
BEGIN
  G : TYPE

  e : G
  i : [G -> G];                   % -- Semikolon
  * : [G,G -> G]                  % -- Infix-Operator

  x,y,z : VAR G

  assoziativ : AXIOM
    (x * y) * z = x * (y * z)

  linksneutral : AXIOM
    e * x = x

  linksinvers : AXIOM
    i(x) * x = e

  invers_assoziativ : THEOREM
    i(x) * (x * y) = y

END gruppe
```

Der PVS-Beweiser (1)

- Die Beweiserkomponente von PVS baut auf einer Implementierung des Sequenzenkalküls auf.
- Darstellung von Sequenzen in PVS:
  Eine Sequenz

  $$A_1, A_2, \ldots, A_n \vdash B_1, \ldots, B_m$$

  wird repräsentiert in der folgenden Form:

```
{-1}   A1
{-2}   A2
...
[-n]   An
   |-------
{1}    B1
...
[m]    Bm
```

Der PVS-Beweiser (2)

- Ein Beweis in PVS entspricht einem Beweisbaum, der von der Wurzel, der zu beweisenden Sequenz, ausgehend konstruiert wird.
- Beweise werden durch Anwendung von Beweiserregeln konstruiert, welche ein oder mehrere Unterziele generieren.
- Eine bestimmte Sequenz (current sequent) ist zu jeder Zeit der Fokus des Beweisprozesses. Auf sie beziehen sich jeweils die Anweisungen an den Beweiser. Wenn ein Beweiskommando mehrere Teilziele erzeugt, wandert der Fokus auf das "erste" Teilziel.
- Teilziele können in beliebiger Reihenfolge bearbeitet werden
- Wird ein Beweisast als bewiesen anerkannt, wandert der Fokus zum nächsten noch offenen Teilziel.
- Falls es kein solches mehr gibt, ist der Beweis abgeschlossen.

Der PVS-Beweiser (3)

- Vor einem Beweis muss die Theorie auf Typkorrektheit hin überprüft werden. Kommando `M-x tc`.
- Cursor auf die zu beweisende Formel, dann `M-x pr`.
- Beweis wird interaktiv in einem zweiten Buffer durchgeführt.
- Bei Anwendung einer Regel können mehrere Beweisziele auftreten. Man sieht jeweils nur das gerade aktuelle. Mit `M-x siblings` kann man die anderen Unterziele ansehen.
- Zum nächsten noch offenen Unterziel gelangt man mit `(postpone)`.
- Mit (undo n) werden die letzten n auf dem aktuellen Beweiszweig getätigten Schritte rückgängig gemacht.
- Vorzeitiges Abbrechen eines Beweises mit `(quit)`.
- Bei nochmaligem Durchführen des Beweises (des evtl. modifizierten Theorems) wird eine rerun-Option angeboten.

Beweiskommandos (1)

- Hilfe zu einzelnen Beweiskommandos: `(help` *<Kommando>*`)`
- Wichtigste Kommandos siehe PVS-Kurzanleitung
- Überblick über alle Kommandos im PVS Prover Guide
- Propositionale Regeln:
  - `(flatten)`: Rückwärtsanwendung der Regeln $\wedge L$,$\vee R$,$\Rightarrow R$
  - `(split)`: Anwendung von $\wedge R$, $\vee L$, $\Rightarrow L$
  - `(case)`: Anwendung der cut-Regel
  - Regeln für die Negation $\neg L$, $\neg R$ werden automatisch angewandt

Beweiskommandos (2)

Quantorenregeln:

- `(inst)`: Anwendung der Regeln $\forall L$, $\exists R$
- `(skolem)`: Anwendung der Regeln $\forall R$, $\exists L$
- Bedingungen an die eingesetzten Terme und Variablen (*variable capture*, *Eigenvariablenbedingung*) werden von PVS automatisch sichergestellt.
- `(inst? &opt` *`fnum[*] subst`*`)`
  - Ohne Parameter: heuristische Instanzierung aller Formeln.
  - Mit Schlüsselwort `:subst` und Liste der Form `(`*`var1 term1 var2 term2 ...`*`)` kann Substitution vorgegeben werden.
- `(skolem! &opt` *`fnum[*]`*`)`: automatische Generierung von Skolemkonstanten.
- `(skosimp*)`: wiederholtes Skolemisieren und disjunktive Vereinfachungen (`flatten`).

Beweiskommandos (3)

PVS bietet etliche weitere Beweiskommandos:

- zur Behandlung von
  - Gleichheiten: `(beta)`, `(replace)`, `(iff)`, ...
  - Definitionen und Lemmata: `(expand)`, `(lemma)`, `(rewrite)` ...
  - bedingten Ausdrücken: `(lift-if)`
- mächtige Beweisprozeduren, wie
  - Entscheidungsprozeduren: `(simplify)`, `(ground)`, `(assert)`,...
  - Induktionsbeweise: `(induct)`, `(induct-and-simplify)`, ...
  - automatisches Vereinfachen von Beweiszielen samt Anwendung von Lemmata und Entscheidungsprozeduren: `(auto-rewrite)`, `(bash)`, `(smash)`, `(grind)`, ...
- Sprache zur Definition eigener Beweisprozeduren

Beweisverwaltung

- PVS speichert Beweise in einer separaten Datei *name*`.prf`
- Mehrere Versionen eines Beweises für eine Formel möglich
- PVS führt Buch über den Beweis-Status einer Formel:
  - untried: es liegt noch kein Beweis für die Formel vor
  - unfinished: Beweis wurde abgebrochen
  - proved – incomplete: Formel ist bewiesen; Beweis hängt aber von noch unbewiesenen Formeln ab.
  - proved – complete: Formel ist bewiesen und ebenso alle in der Beweiskette auftretenden Formeln
  - unchecked: Formel ist bewiesen; Theorie wurde jedoch zwischenzeitlich verändert
- Statusabfrage mit `M-x status-proof-theory`
- Abspeichern von Theorien mit Beweisen in einer Dump-Datei: `M-x dump-pvs-files`
  - Übungsaufgaben bitte so abgeben!
  - Auspacken einer Dump-Datei: `M-x undump-pvs-files`